Report Number: WUCS-92-22

1992-01-01

The DIM system Plan Recognition in Spoken Dialogues

Umesh Berry and Michael Groner

Although speech recognition has improved significantly in recent years, it use in commercial environments has been limited by the inability of systems to model conventional co-operative dialogue. Any natural language systems for co-operative dialogue should at least be able to account for the many types of sub-dialogues that occur in conversations. Plan-based techniques provide a computationally powerful method for handling such dialogues. We discuss in this report our first attempt at building a plan-recognizer for an overall speech-driven system which interacts via a naturally restricted subset of English in a limited domain.
... Read complete abstract on page 2.

The DIM system Plan Recognition in Spoken Dialogues

Umesh Berry and Michael Groner

Complete Abstract:

Although speech recognition has improved significantly in recent years, it use in commercial environments has been limited by the inability of systems to model conventional co-operative dialogue. Any natural language systems for co-operative dialogue should at least be able to account for the many types of sub-dialogues that occur in conversations. Plan-based techniques provide a computationally powerful method for handling such dialogues. We discuss in this report our first attempt at building a plan-recognizer for an overall speech-driven system which interacts via a naturally restricted subset of English in a limited domain.

The DIM system :

Plan Recognition in Spoken Dialogues

Umesh Berry and Michael Groner

WUCS-92-22

July 1, 1992

Department of Computer Science
Campus Box 1045
Washington University
One Brookings Drive
St. Louis, MO 63130.

Abstract

Although speech recognition has improved significantly in recent years, its use in commercial environments has been limited by the inability of systems to model conventional co-operative dialogue. Any natural language system for co-operative dialogue should at least be able to account for the many types of sub-dialogues that occur in conversations. Plan-based techniques provide a computationally powerful method for handling such dialogues. We discuss in this report our first attempt at building a plan-recognizer for an overall speech-driven system which interacts via a naturally restricted subset of English in a limited domain.

Contents

1. Introduction

We are currently working on developing a robust, speech-driven dialogue system which interacts via naturally restricted spoken sentences in a chosen application domain. In this report, we discuss the plan recognition component, which forms the core of the overall system. The domain that we are dealing with is the management of custom calling features provided by Southwestern Bell Telephone (SWBT). This domain is task-oriented and information-seeking.

One promising approach to analyzing such dialogues has involved modelling of plans of the speakers in the task domain. Researchers have shown that plan-based dialogue management is a crucial part of any natural language processing system (Litman and Allen, 1987; Lambert and Carberry, 1991). They have differentiated between the various ways that an utterance can relate to a plan structure representing a topic. Specifically, utterances may change or clarify an existing plan, introduce a new plan, continue an existing plan or accompany the execution of the next step in a plan. We have built the Plan Recognizer (PR) based on the theory developed by Litman and Allen. The theory integrates knowledge about discourse, allowing for a wide variety of sub-dialogues while maintaining the computational advantages of a plan-based approach.

The techniques of plan-based dialogue processing suggested by Litman and Allen were based on the assumption that input to the system is text-based. We believe that the same model can be used effectively for speech input also (Johnstone and Balentine, 1992). Before we began work on the PR, we wanted to have a good idea of the kind of situations we would encounter in our domain. We did not want to build the system on the basis of dialogues that were contrived. So, during the summer of 1991, we conducted a "seed" experiment and collected dialogues of users interacting with a mock natural language telephone interface (Balentine, Berry and Johnstone, 1992). The corpus of inputs obtained served as a guideline for building both the Natural Language Processor, NLP, (Balentine, Johnstone and Mathias, 1992) and the PR.

In the next section we discuss the theoretical issues involved in dialogue management. In section three we discuss the model for the plan recognizer. In section four we illustrate the working of the system with a few examples from our domain. Next, we discuss the implementation and conclude with an analysis of the current system. The work done is in no way complete, but this first attempt has been invaluable for us in realizing the work that needs to be done to make the system perform well in a real setting.

2. Theoretical Issues

Previous research work has been primarily on modelling human-human dialogue. It is possible that the issues involved in modelling human-machine dialogue may not be the same as that for human-human dialogue.

Any dialogue management system should co-ordinate the activities of the many parts of the complete system to obtain efficient human-machine dialogue. The behaviours that we wish the PR to exhibit are similar to those shown by the dialogue control algorithm built by (Smith, Hipp and Biermann, 1992). These are :

2.1 Convergence to a goal

For an efficient human-machine interaction, the machine has to play the role of an intelligent agent. Both the conversants should understand the purpose of the interaction and contribute to achieve this goal.

2.2 Subdialogues

It is often the case that in a dialogue, the conversants do not keep the focus fixed. A dialogue is usually segmented into utterance sequences, subdialogues, that are individually aimed at achieving relevant subgoals. The dialogue manager should recognize these subdialogues and respond appropriately.

2.3 Modelling user capabilities and beliefs

There are numerous instances where an utterance's interpretation hinges on the beliefs of the person who uttered it. For example, "Can you speak Spanish?", could either be a request from the speaker to the hearer to speak in Spanish or it could be a question (Hinkelman, 1990). The utterance would be a request if the speaker believes that the hearer can speak Spanish, and would be a question if the speaker doesn't know whether the hearer can speak Spanish or not. Another reason why such a model is needed is to have an idea of what needs to be said to the user to enable the user to function effectively. Some things need not be said for the user may already know them.

2.4 Change of initiative

For maximum efficiency, the dialogue manager should realize when to take the initiative and when to let the user have the control. The dialogue manager should take the initiative when it realizes that the user is incapable of reaching the required goal, that is when it feels that it is the more competent partner in the dialogue. On the other hand, when a user knows how to achieve the goal, the dialogue manager should let them have the control for faster execution.

2.5 Expectation of utterances

In human-human dialogue, people have expectations for utterances based on what was previously said. The system should also have this capability. This would enable the system to correctly interpret an utterance which was misinterpreted at the front-end by the speech-recognizer, that is, it would make the system very robust. Also, expectations would help the system to keep track of the dialogue as it jumps from one subdialogue to another.

As has been observed by previous researchers, systems capable of all these behaviors are rare. It is difficult to integrate all the pieces in a coherent way. Our current implementation exhibits the first two and the last behaviour and we are working on incorporating user models and the change of initiative capability in the system.

3. The Plan Recognizer

3.1 Overview

In order to differentiate between the various ways that an utterance can relate to the current state of the dialogue, plans are classified into two categories : discourse plans and domain plans. Discourse plans model how an utterance relates to the current discourse topic (for example. it might clarify an existing plan, it might correct some step of an existing plan etc.).They are domain independent. Domain plans are the set of plans that are used to model the topic (in our case, the management of custom calling features like call-forwarding, call-waiting, speed-dialling etc.). Such a plan-based formulation enables us to use the same techniques of plan recognition for both discourse plans and domain plans. Apart from being a well-defined method of representation, the formulation is computationally efficient.

It has been observed by previous researchers (Polanyi and Scha, 1983; Reichman-Adar, 1984) that in limited domains, conversations follow a stack-like discipline, i.e. during a conversation new topics might be introduced at any time, temporarily suspending the current topic, and upon completion the previous topic is resumed. So at any stage, the current discourse topic is represented by a stack of plans. In the simplest case, the plan-stack consists of a domain plan at the bottom and a discourse plan at the top. In clarification sub-dialogues, normally there are two discourse plans (the lower one being the one that is clarified) at the top and the domain plan at the bottom. Similarly for topic-shifts there may be more than one domain and discourse plan on the plan-stack. We shall illustrate such instances in section four, where we'll consider some examples.

The basic idea is that each new utterance must be classified as part of a new discourse plan, which itself is related to a domain plan. In the simplest case, there is one incomplete domain plan at the bottom, the completed discourse plan at the top (representing the last interaction) is popped, and the new discourse plan pushed in it's place. In other cases, the old domain plan and discourse plans remain on the stack and additional domain and discourse plans are pushed on top of them. The PR attempts to classify each utterance in terms of a discourse plan.

3.2 The Plan-stack

At any instant, the stack represents what the system believes is the state of the joint plan. During a dialogue, the stack consists of executing, suspended or completed plans. Once the plan recognition process is completed, the observed action is always is in the plan that is on top of the stack. Even when the system believes the top plan has successfully completed, it can't be popped unless there is an acknowledgement, explicit or implicit (for example. execution of a step in a plan that is not on the top). The top of the stack is either an executing or a just executed plan. The rest of the stack consists of other suspended plans.

The stack discipline might be too strict for many naturally occuring conversations where a previous topic may not be resumed. Based on experimental evidence, we believe that this model is sufficient for human-machine dialogues in a limited domain (Balentine, Berry and Johnstone, 1992).

3.3 Representation of the Plans

The representation of plans is similar to that employed in many models of planning (for example. STRIPS (Fikes and Nillson, 1971)). Every plan has a header, a parameterized action description that names the plan. Plans may contain prerequisites, add lists, delete lists, decompositions and constraints. Prerequisites are conditions that need to hold (or be made to hold) before the plan can be executed. Add lists contain conditions that will hold after the plan has been successfully completed. Delete lists contain conditions that will no longer hold after the plan has bees successfully completed. Decompositions are the steps in the plan (they may be plans themselves). They enable hierarchial planning. Constraints are similar to prerequisites except that the planner never attempts to achieve them in case they are false. A library of plan schemas is used to represent knowledge about typical speaker plans. These plan schemas will be used both for plan recognition as well as plan generation.

As an example, the following schema represents the speaker's plan to get help from the hearer on some feature of the domain :

HEADER	:	gethelp(S, H, F)
DECOMPOSITION	:	help(H, S, F)
ADDLIST	:	complete(gethelp(S, H, F)), last(help(H,S, F), gethelp(S,H, F))
DELETE LIST	:	next(help(H, S, F), gethelp(S, H, F))

Throughout this report we shall adopt the following conventions :

1. Capital letters in predicate arguments represent uninstantiated variables.
2. S, H, F, A, P stand for Speaker, Hearer, Feature, Action, Plan respectively.
3. The first argument of any plan represents the agent.

3.4 Discourse Plans

Discourse plans model the knowledge of the conversational process using the plan-based framework. These are domain independent and relate every utterance to domain plans. The schema for the discourse plans is similar to that for domain plans, the only difference being that discourse plans take on other plans as arguments and are thus meta-plans. Thus domain plans model the contents of the domain and discourse plans model the manipulations of the topic in a dialogue.

In order to include plans as objects, Litman and Allen have defined some predicates that have plans as their arguments. These are :

step(A, P) :	asserts that action A (possibly a plan) is a step in plan P.
parameter(Par,P) :	asserts that term Par is a parameter of the action description in the header of plan P. For example. parameter(F, gethelp(S, H, F)) is true.
complete(P) :	asserts that plan P is complete.
next(A,P) :	asserts that action A is the next step to be performed in plan P.
last(A, P) :	asserts that action A was the last step to be successfully performed in plan P.

The set of discourse plans are divided roughly into four categories : the continuation class, the clarification/correction class, the topic-shift class and the conclusion class. We give here an informal description of the plans. Formal descriptions of each plan are given in Appendix A.1.

3.4.1 The continuation class

CONTINUE-PLAN :	The agent executes the next action in the top-most non-complete plan on the stack.
TRACK-PLAN :	The agent talks about the execution of the next action in the top-most non-complete plan on the stack. These include utterances which accompany some action.

3.4.2 The clarification/correction class

IDENTIFY-PARAMETER : involves providing a suitable description for a term instantiating a parameter of an action such that the hearer is then able to execute the action.

CORRECT-PLAN : involves specifying correction of a plan in order to ensure its success after unexpected events at runtime.

3.4.3 The topic-shift class

INTRODUCE-PLAN : This introduces a new plan for discussion that is not part of the previous topic of conversation (could be explicitly or implicitly triggered).

3.4.4 The conclusion class

END-PLAN : indicates the completion of a plan successfully and triggers the end of dialogue if there are no more non-complete plans.

3.5 Domain Plans

Domain plans model the contents of the topic and are kept separate from discourse plans. So by using the same set of discourse plans, the PR could be used for another application by just changing the set of domain plans appropriately. An example of a domain plan can be found in section 2.3. A complete formal description of the domain plans is given in Appendix A.2.

3.6 Speech Acts

The speech acts (Searle, 1969) are modelled in the plan-based framework (Allen and Perrault, 1980) with the exception that indirect speech acts are modelled in the decompositions. For a complete definition of speech acts in the plan-based framework, see Appendix B. An example of an indirect speech act is someone saying "I want to forward my calls" which has a surface form corresponding to an inform but could be a request to forward the calls. The fourth decomposition of the request speech act takes this into account. The speech act recognition is based on differences of discourse context. So, whether the above utterance is an inform or a request is decided by the discourse context. The underlying assumption is that utterances are heard and understood correctly by the listener as they are uttered and it is expected that they will be so understood. Also, at the moment we have a single speech act associated with each utterance. These assumptions are too strong to be able to handle many types of conversations people have. (Traum, 1991) has proposed a hierarchial classification scheme for speech acts and has demonstrated how some of the current limitations can be overcome. We will try to incorporate these ideas into our speech act framework in the near future.

3.7 Plan recognition

The aim of the plan recognition process is to infer a domain plan underlying the production of every utterance and at the same time deduce the utterance's relationship with this domain plan. The PR has at it's disposal the libraries for the domain and discourse plans, the speech act definitions and the current plan stack. Input to the PR is the surface-form

corresponding to the input utterance. Each such input is classified as part of the execution of a discourse plan. The final output of the PR is a surface-form which constitutes an appropriate response. In case there was an action requested, the PR sends the appropriate signals to ensure that the requested action is performed. The plan-stack is updated throughout the course of the dialogue and at any instant represents the system's current view of the conversation.

The PR's task is to find a sequence of instantiations of plan schemas, each one containing the previous ones in it's decomposition (decomposition chaining), that connects the surface-form produced by the Natural Language Processor to a discourse plan. Since each discourse plan has another plan (domain or discourse) as an argument, every time the surface-form is connected to a discourse plan, this new plan is also introduced. This plan could be new (as in topic-shifts) or could already be present on the stack (as in continuations, clarifications). The constraints associated with each plan ensure that incorrect domain and discourse plans are never considered.

Once a set of plans is recognized (that is, the domain plan has been reached), all plans are expanded top-down and placed on the stack. The original discourse plan is on the top, each discourse plan refers to the plan below it and the domain plan is at the bottom. This entire chaining process is illustrated in section four using examples from our domain.

The above recognition process is constrained further by applying coherence heuristics, which prefer the most linguistically coherent continuation of the dialogue represented by the plan-stack. Utterances are viewed as possible continuations first. If an utterance can be seen as a continuation of an existing plan, then no other interpretation is tried. Clarification or correction plans are attempted next. Topic-shift plans are attempted if no continuations or clarifications can be found on the plan-stack.

These are not the only heuristics that are applied. Even though the PR might suggest a plan as being linguistically coherent, the plan might not be rational. For example it's effects might already be true or constraints might not be satisfiable. These heuristics, referred to as plan-based heuristics by Litman ands Allen, eliminate irrational plans from consideration.

4. Examples

For the purposes of illustration of the reasoning process of the PR, we shall analyze in detail in Example 1 a very simple dialogue involving no corrections, clarifications or topic-shifts. Example 2 illustrates clarifications. Examples 3 and 4 show how the same utterance can mean two different things in different discourse contexts. Finally Example 5 illustrates corrections. In all the examples U and S will represent the user and the system respectively.

4.1 Example 1

U : Help me with dialing.
S : To dial a number ....
U : Thanks.

The surface-form corresponding to the first utterance by the user is :
surface-request(u, s, help(s, u, forward))

The stack is initially empty and using coherence heuristics the introduce-plan discourse plan is found via decomposition chaining.

```
plan1  :      introduce-plan(u, s, h1, P)
                          |
                          |
                    request(u, s, h1)
                          |
                          |
      surface-request(u, s, h1 : help(s, u, forward))
```

At this point other possibilities for discourse plans (correct-plan, continue-plan etc.) are not considered because their constraints are violated. The next thing the PR does is ensures that plan1 is reasonable, that is it's effects are not already true. Since plan1 is reasonable, the PR next attempts to get a binding for the uninstantiated plan P in plan1 through a process called constraint satisfaction. In this case the constraint is that h1 should be a step in P. The only possible candidate plan is :

```
plan2  :      gethelp(u, s, forward)
                    |
                    |
              h1 : help(s, u, forward)
```

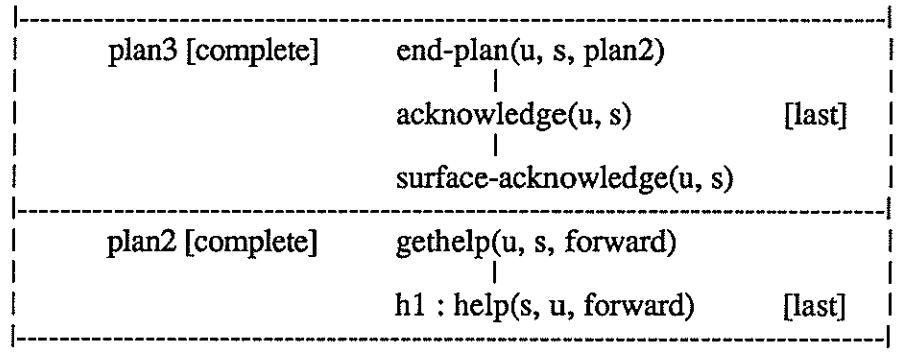
Again the PR ensures that plan2 is reasonable. Since a domain plan has been reached, the chaining is stopped and the effects of the recognized plans are asserted. The plans are expanded top-down and placed on the plan-stack. At this instant the plan-stack looks something like this :

```
|-------------------------------------------------------------------------|
|     plan1 [complete]     introduce-plan(u, s, h1, plan2)                |
|                                       |                                 |
|                                 request(u, s, h1)      [last]           |
|                                       |                                 |
|                          surface-request(u, s, h1)                      |
|-------------------------------------------------------------------------|
|     plan2                gethelp(u, s, forward)                         |
|                                  |                                      |
|                          h1 : help(s, u, forward)      [next]           |
|-------------------------------------------------------------------------|
```

It is now the system's turn in the conversation and the system finds that the top of the stack is a complete plan so it pops it off and reaches the top-most non-complete plan which in this case is plan2. Then it executes the next step to be performed in plan2 which in this case is providing help and marks this as the last action performed leading to this plan-stack :

```
|-------------------------------------------------------------------------|
|     plan2 [complete]     gethelp(u, s, forward)                         |
|                                  |                                      |
|                          h1 : help(s, u, forward)      [last]           |
|-------------------------------------------------------------------------|
```

The user then thanks the system and the utterance has the following surface-form : surface-acknowledge(u, s).The discourse plan to which this utterance links is end-plan. Note however that in this case the plan to which the end-plan refers to can be found on the plan-stack itself and so no constraint satisfaction to get a new domain plan is required. The plan-stack at this stage signals an end of conversation and looks like this :

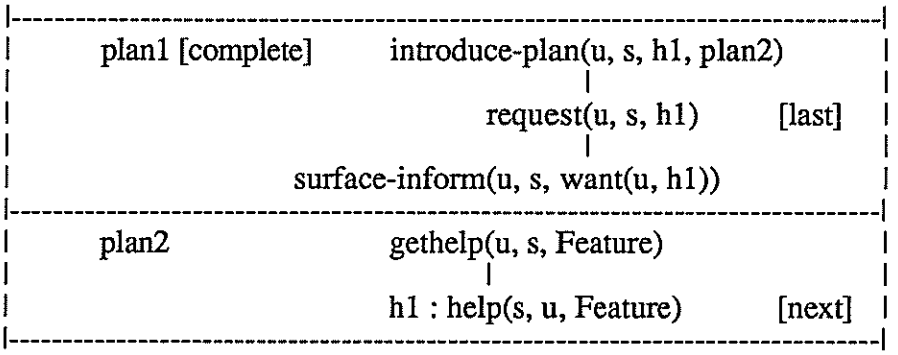


4.2 Example 2

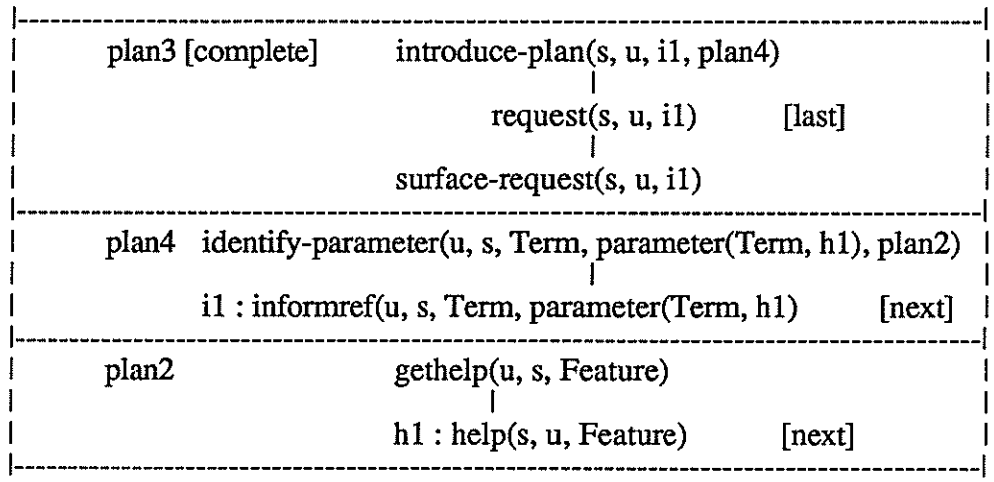
U: I want help.
S: Please say what feature you want help with.
U: Call-forwarding.
S: With call-forwarding you can ....
U: Thanks.

The surface-form of the first utterance by the user is :
surface-inform(u, s, want(u, help(s, u, Feature)))

This surface-form is a possible decomposition for an inform or a request speech act. At the moment the inform speech act does not lead to any discourse plans, that is we are treating all informs as requests. In the near future we plan to build a user model which incorporates the beliefs of the user. This would be useful in differentiating between informs and requests. Thus the speech act underlying the utterance is : request(u, s, help(s, u, Feature)). This leads to the introduce-plan discourse plan. Constraint satisfaction as before binds the plan being introduced to gethelp(u, s, Feature). The plan stack after the utterance is

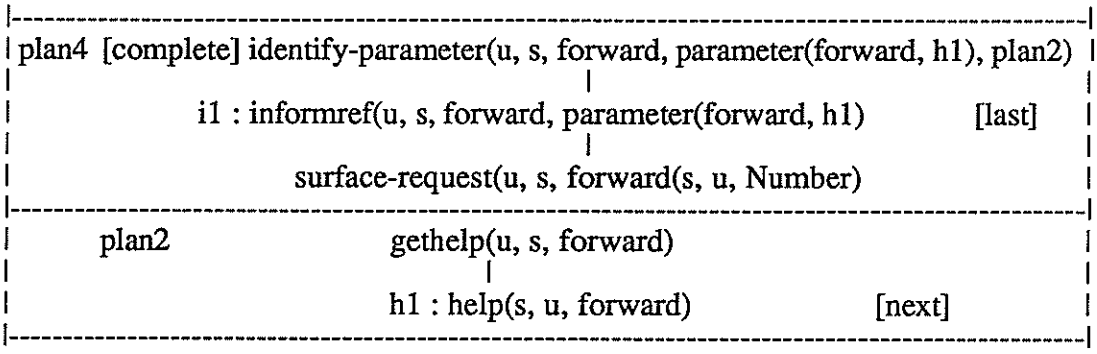


The system pops the top complete plan and reaches plan2, the top-most non-complete plan and finds that one of the parameters of the plan is unspecified (the feature that the user wants help with). So the PR generates a plan to obtain the missing piece of information. After making a request to the user to say what feature the user wants help with, the plan stack is :



The surface-form of the user's next utterance is :
surface-request(u, s, forward(s, u, Number))

There are two possible speech acts associated with this utterance. Either the user is requesting the system to forward the user's calls to some as yet unspecified number or the user is supplying some missing information (an informref). The discourse context establishes the expectation for the next utterance to be an informref and so this interpretation is preferred. At this point the plan stack is :



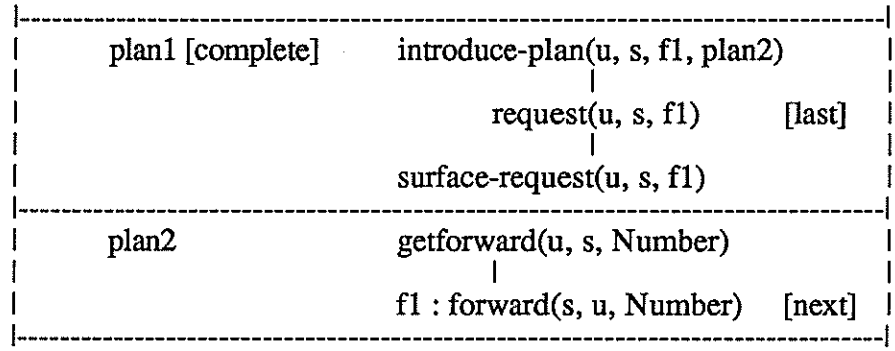
The rest of the execution is as before where the system pops off the top plan, gets to the top-most non-complete plan and this time it can be executed since all the parameters are specified.

4.3 Example 3

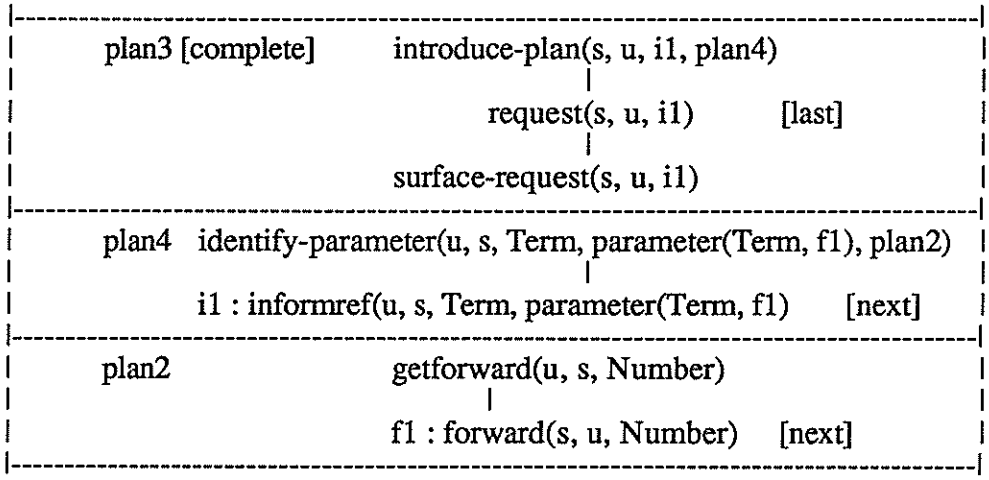
U: Call-forwarding.
S: Please say the number that you want your calls forwarded to.
U: 345-8721.
S: Calls are forwarded to 345-8721.
U: Thanks.

As in example 2, the surface-form for the first utterance by the user is :
surface-request(u, s, forward(s, u, Number))

This time the request interpretation is preferred as there is no expectation from the discourse context. The plan stack is :

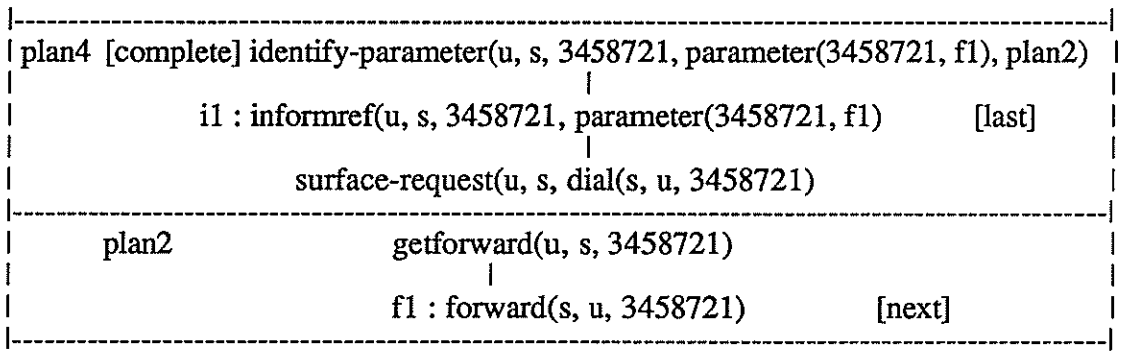


As before when the PR attempts to perform the next action, it finds that the telephone number is missing and requests the user for the same. The plan stack after the request from the user is :



The user responds by uttering the telephone number whose surface-form is :
surface-request(u, s, dial(s, u, 3458721)).

Again, of the two possible speech acts, request or informref, the latter is preferred due to the expectation from the discourse context. At this stage the plan-stack looks like this :

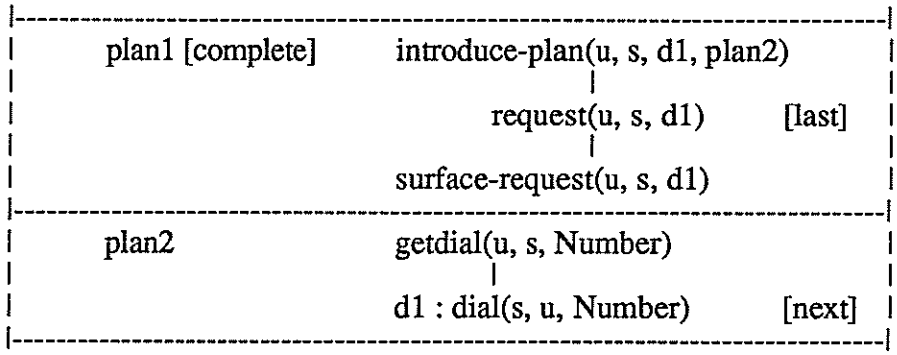


Now that the missing piece of information has been supplied by the user, the system can go ahead and execute the next step in the top-most non-complete plan. The remaining dialogue is handled in a manner similar to previous examples.

4.4 Example 4

U: 345-8721.
S: Now dialing 345-8721.
U: Thanks.

In this example the user's first utterance is the same as the second one in the previous example, but the PR is able to recognize that in this case the user intends the system to dial the number because there is no discourse context to believe otherwise. The analysis of this dialogue is similar to the first example since all the necessary information is supplied by the user in the first utterance. We shall only give the plan-stack after the user's first utterance. The rest of the dialogue is similar in structure to example three.



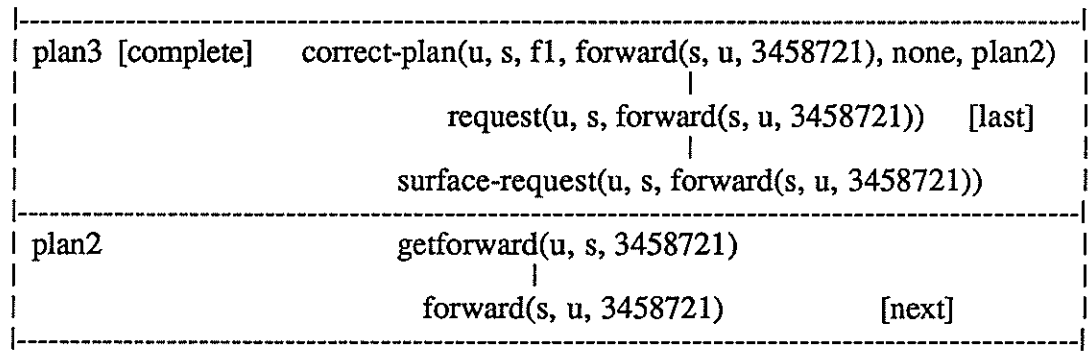
4.5 Example 5

U: Forward my calls to 345-8725.
S: Calls are being forwarded to 345-8725.
U: No, forward my calls to 345-8721.
S: Calls are being forwarded to 345-8721.
U: Thanks.

The analysis for the user's and the system's first utterance is as before. The surface-form of the user's second utterance is :

surface-request(u, s, forward(s, u, 345-8721))

At this point there is no expected action from the user (except an acknowledgement), but the surface-form is a request for some action. The decomposition chaining leads to the correct-plan discourse plan and the plan-stack after the recognition of the correct-plan is :



The rest of the execution steps are as before.

5. Implementation

The algorithm that we used to implement the plan recognizer is :

Repeat

1) Use coherence heuristics to choose a discourse plan via decomposition chaining.
   The order of preference is :
   i) continuation plans
   ii) clarification/correction plans
   iii) topic-shift plans
   iv) conclusion plans
2) Apply plan-based heuristics
   i) Check to see that effects are not already true.
   ii) Use constraint satisfaction to relate discourse plans to domain plans.
3) If there is more than one domain plan or a top level domain plan has been reached go to step 4
   Else recursively chain on the current plan.
4) Expand all plans top down and push on plan-stack.
5) Assert all effects of recognized plans into world model.
6) Perform next action.

Until end of dialogue is reached.

This algorithm has been implemented in Quintus Prolog and is able to handle all the dialogues that we discussed in section four. The code is given in Appendix C.

6. Limitations and Future Work

We have used discourse and domain plans to model the structure of natural language conversations in a limited domain. The assumption that we have made is that in limited domains like the one that we are dealing with, the conversation follows a stack-like

discipline. The discourse and domain plans help us recognize and generate plans referring to the task at hand. The plan-stack is used to organize these plans in an appropriate way. The implementation allows us to handle straightforward examples, but there are still situations that the PR does not cover yet.

As mentioned in section two, there are some capabilities which an intelligent agent must have to participate in a dialogue effectively. Currently the system does not have a model for the user's beliefs and capabilities. Consequently, it has no way of telling whether an utterance like "Can you forward my calls?" is a yes-no question or a request for action. At present we are working on having a complete help sub-system, which should enable the system to realize when a user needs help without the user explicitly stating so. This would be a part of incorporating the initiative-switching capability in the system.

In a speech-driven dialogue system, it is essential that the system be capable of realizing when to take and when to release the turn in the dialogue. Also, it should know how to do the same. We are working on modelling turns as utterance acts, similar to requests, informs etc. We intend to modify our speech-act theory so that it can model utterances which accomplish multiple utterance acts and at the same time account for speech acts that are accomplished after multiple utterances. We haven't included in the system the capability to answer questions about the state of things in our domain. For example, at present the PR doesn't account for utterances like "Is call waiting on?", which is a query about the status of some feature.

As is evident, the system is in a nascent stage now and a lot of work needs to be done to make it perform well in a real setting. Once we have a working system, it could be used effectively in any other limited domain.

7. Acknowledgements

We would like to thank Anne Johnstone for her help and encouragement while working on the project and in writing this report.

APPENDIX A.1

Discourse Plans

HEADER	introduce-plan(S, H, A, P)
(Takes a plan of the speaker that involves the hearer and introduces it into the conversation. This may be done via a request from the speaker that the hearer perform some action in the plan.)	
DECOMPOSITION	request(S, H, A)
ADDLIST	complete(introduce-plan(S, H, A, P)) next(A, P) last(request(S,H,A), introduce-plan(S, H, A, P))
CONSTRAINTS	step(A, P)

HEADER	identify-parameter(S, H, Par, A, P)
(Describes a parameter of an action that is a step of a plan. This is done through the linguistic act informref)	
DECOMPOSITION	informref(S, H, Par, parameter(Par, P))
ADDLIST	complete(identify-parameter(S, H, Par, A, P)) last(informref(S, H, Par, parameter(Par, P)),identify-parameter(S, H, Par, A, P))
CONSTRAINTS	step(A, P)

HEADER	correct-plan(S, H, Last, New, Next, P)
(Allows unexpected events at run time to be dealt with)	
PREREQUISITES	last(Last, P)
DECOMPOSITION	request(S, H, New)
ADDLIST	complete(correct-plan(S, H, Last, New, Next, P)) next(New, P) last(request(S, H, New), correct-plan(S, H, Last, New, Next, P))
CONSTRAINTS	top-plan(P) next(Next, P)

HEADER (Signals the conclusion of the plan)	end-plan(S, H, P)
DECOMPOSITION	acknowledge(S, H)
ADDLIST	complete(end-plan(S, H, P)) last(acknowledge(S,H), endplan(S, H, P))
CONSTRAINTS	top-plan(P)

APPENDIX A.2

Domain Plans

HEADER	gethelp(S, H, F)
(Speaker wants help from the Hearer on some Feature)	
DECOMPOSITION	help(H, S, F)
ADDLIST	complete(gethelp(S, H, F)) last(help(H, S, F), gethelp(S, H, F))
DELETELIST	next(help(S, H, F))

HEADER	getcancel(S, H, F)
(Speaker wants the Hearer to cancel some Feature)	
DECOMPOSITION	cancel(H, S, F)
ADDLIST	complete(getcancel(S, H, F)) last(cancel(H, S, F), getcancel(S, H, F))
DELETELIST	next(cancel(S, H, F))

HEADER	getforward(S, H, Number)
(Speaker wants the Hearer to forward calls to some Number)	
DECOMPOSITION	forward(H, S, Number)
ADDLIST	complete(getforward(S, H, Number)) last(forward(H, S, Number), getforward(S, H, Number))
DELETELIST	next(forward(S, H, Number))

HEADER getremforward(S, H, From, To)
(Speaker wants the Hearer to forward calls from some number From to some number To)

DECOMPOSITION remforward(H, S, From, To)

ADDLIST complete(getremforward(S, H, From, To))
last(remforward(H, S, From, To), getremforward(S, H, From, To))

DELETELIST next(remforward(S, H, From, To))

HEADER getdial(S, H, NumberorCode)
(Speaker wants the Hearer to dial the number or code NumberorCode)

DECOMPOSITION dial(H, S, NumberorCode)

ADDLIST complete(getdial(S, H, NumberorCode))
last(dial(H, S, NumberorCode), getdial(S, H, NumberorCode))

DELETELIST next(dial(S, H, NumberorCode))

HEADER getchange(S, H, Number, O, N)
(Speaker wants the Hearer to change the code for Number from O to N)

DECOMPOSITION change(H, S, Number, O, N)

ADDLIST complete(getchange(S, H, Number, O, N))
last(change(H, S, Number, O, N), getchange(S, H, Number, O, N))

DELETELIST next(change(S, H, Number, O, N))

HEADER getadd(S, H, Number, C)
(Speaker wants the Hearer to add the code C for Number to the speed dial directory)

DECOMPOSITION add(H, S, Number, C)

ADDLIST complete(getadd(S, H, Number, C))
last(add(H, S, Number, C), getadd(S, H, Number, C))

DELETELIST next(add(S, H, Number, C))

HEADER getdelete(S, H, NumberorC)
(Speaker wants the Hearer to delete the number or the code NumberorC from the speed dial directory)

DECOMPOSITION delete(H, S, NumberorC)

ADDLIST complete(getdelete(S, H, NumberorC))
last(delete(H, S, NumberorC), getdelete(S, H, NumberorC))

DELETELIST next(delete(S, H, NumberorC))

HEADER hearcodes(S, H)
(Speaker wants the hear the codes stored in the speed dial directory from the Hearer)

DECOMPOSITION showcodes(H, S)

ADDLIST complete(hearcodes(S, H))
last(showcodes(H, S), hearcodes(S, H))

DELETELIST next(showcodes(S, H))

APPENDIX B

Speech-act Definitions

HEADER request(S, H, A)
(The speaker requests the hearer for action A)

DECOMPOSITION
1. surface-request(S, H, A)
2. surface-yn-question(S, H, cando(H, A))
3. surface-inform(S, H, not(cando(S,A)))
4. surface-inform(S, H, want(S, A))

HEADER inform(S, H, Prop)
(The speaker informs the hearer that proposition P is true)

PREREQUISITES know(S, Prop)

DECOMPOSITION surface-inform(S, H, Prop)

ADDLIST
know(H, Prop)
know(H, know(S, Prop))

HEADER informref(S, H, Term, Prop)
(The speaker tells the hearer some proposition that is true of parameter Term.)

DECOMPOSITION
1. surface-informref(S, H, Term, Prop)
2. surface-request(S, H, A)

CONSTRAINTS parameter(Term, Prop)

HEADER informif(S, H, Prop, TorF)
(The speaker tells the hearer whether some proposition is true or not.)

DECOMPOSITION 1. surface-informif(S, H, Prop, TorF)

HEADER yn-question(S, H, Prop)
(The speaker asks the hearer whether some proposition is true or not.)

DECOMPOSITION 1. surface-yn-question(S, H, Prop)

CONSTRAINTS not(know(S, Prop))

HEADER acknowledge(S, H)
(The speaker acknowledges the success of the interaction to the hearer.)

DECOMPOSITION 1. surface-acknowledge(S, H)

APPENDIX C

Code-listing

```
/*************************************************************************
                        Plan-based Dialogue Manager
                        ---------------------------
*************************************************************************/
:- ensure_loaded(library(basics)).
:- consult('speech-act-library').
:- consult('discourse-plan-library').
:- consult('domain-plan-library').
:- consult('database').
:- consult('input5').

:- dynamic possible_speechacts/1.
:- dynamic plans/1.
:- dynamic list_of_chained_plans/1.
:- dynamic plan_stack/1.
/*************************************************************************
        possible_speechacts(A)    : A is a list of possible speechacts
                                    underlying every utterance
        plans(A)                  : A is a list of possible discourse plans
                                    underlying every utterance
        list_of_chained_plans(A)  : A is a list of recognized plans after a
                                    top-level domain plan has been reached
        plan_stack(A)             : A is the current plan stack
*************************************************************************/
possible_speechacts([]).
plans([]).
list_of_chained_plans([]).
plan_stack([]).

/*************************************************************************
                            run : the main clause which processes utterances
                                  until end of dialogue is signalled
                       form(A) : A is the surface form for utterance
                       restart : makes sure that we start afresh for each
                                  utterance
       recognize_speechacts(A) : finds the possible speech acts underlying
                                  surface-form A
apply_plan_based_heuristics(A) : applies plan based heuristics on the list
                                  of possible speech acts to eliminate some
        decomposition_chain(A) : decomposition chains from the speech acts A
                                  until a top level domain plan is reached
       update_chained_plans(A) : links the plans that have been recognized A
                                  so that they can be pushed on the plan
                                  stack
             asserteffects(A) : asserts the effects of the recognized chain
                                  of plans A
            performnextaction : simulates system's turn
*************************************************************************/
run :-
        testforend,!,nl,nl.
```

```
run :-
        form(InputList),
        retract(form(InputList)),
        restart,
        recognize_speechacts(InputList),
        possible_speechacts(Initial_speech_acts),
        apply_plan_based_heuristics(Initial_speech_acts),
        possible_speechacts(Final_speech_acts),
        decomposition_chain(Final_speech_acts),
        plans(Final_plans),
        update_chained_plans(Final_plans),
        list_of_chained_plans([Chained_plans]),
        push_to_plan_stack(Chained_plans),
        asserteffects(Chained_plans),
        performnextaction,
        run.

recognize_speechacts(Input) :-
        decomp(Speech_act,Input),
        add_to_speech_acts(Speech_act),
        fail.
recognize_speechacts(_).

apply_plan_based_heuristics([First_speech_act|_]) :-
        next(First_speech_act,Plan),!,
        retract(possible_speechacts(_)),
        assert(possible_speechacts([First_speech_act])),
        pop_until(Plan).
apply_plan_based_heuristics([_|Remaining_speech_acts]) :-
        apply_plan_based_heuristics(Remaining_speech_acts).
apply_plan_based_heuristics([]).

decomposition_chain([First_speech_act|Remaining_speech_acts]) :-
        decomp(Plan,First_speech_act),
        reasonable(Plan),
        add_to_plans(Plan),
        decomposition_chain(Remaining_speech_acts).
decomposition_chain([_|Remaining_speech_acts]) :-
        decomposition_chain(Remaining_speech_acts).
decomposition_chain([]).


reasonable(Plan) :-
        present_on_stack(Plan),!.
reasonable(Plan) :-
        addlist(Plan,Effects),
        nottrue(Effects).

present_on_stack(Plan) :-
        plan_stack(Plan_stack),
        member(Plan,Plan_stack).

asserteffects([First_plan|_]) :-
        First_plan=correct_plan(_,_,_,_,_,Plan),!,
        addlist(First_plan,Effect),
```

```
        execute(Effect),
        retract(complete(Plan)).

asserteffects([First_plan|_]) :-
        addlist(First_plan,Effect),
        execute(Effect).

performnextaction :-
        plan_stack(Plan_stack),
        all_complete(Plan_stack),!,
        write('s : surface_acknowledge(s,u)'),nl.
performnextaction :-
        get_non_complete_plan(Plan),
        next(Action,Plan),
        Action =.. [_,_,_|Arguments],
        check_no_unknowns(Arguments),!,
        perform(Action,Plan).
performnextaction :-
        plan_stack([Top_plan|Remaining_plans]),
        next(Action,Top_plan),
        retract(plan_stack(_)),
        assert(plan_stack(Remaining_plans)),
        Action =.. [_|Arguments],
        get_unknown(Arguments,Unknown),
        I1 = informref(u,s,Unknown,parameter(Unknown,Top_plan)),
        P1 = identify_parameter(u,s,Unknown,Action,Top_plan),
        push_to_plan_stack([introduce_plan(s,u,I1,P1),P1,Top_plan]),
        assert(last(request(s,u,I1),introduce_plan(s,u,I1,P1))),
        assert(complete(introduce_plan(s,u,I1,P1))),
        assert(next(I1,P1)),
        write('s : surface_request(s,u,'),write(I1),write(')'),nl,nl.

get_non_complete_plan(Plan) :-
        plan_stack([Top_plan|Remaining_plans]),
        complete(Top_plan),
        retract(complete(Top_plan)),
        retractall(next(_,Top_plan)),
        retractall(last(_,Top_plan)),
        retract(plan_stack(_)),
        assert(plan_stack(Remaining_plans)),
        get_non_complete_plan(Plan).
get_non_complete_plan(Plan) :-
        plan_stack([Plan|_]),!.

update_chained_plans([First_chained_plan|Remaining_chained_plans]) :-
        First_chained_plan=introduce_plan(_,_,_,Plan),
        append([First_chained_plan],[Plan],Result),
        list_of_chained_plans(Initial_chained_plans),
        append([Result],Initial_chained_plans,Final_chained_plans),
        retract(list_of_chained_plans(_)),
        assert(list_of_chained_plans(Final_chained_plans)),
        update_chained_plans(Remaining_chained_plans).
update_chained_plans([First_chained_plan|Remaining_chained_plans]) :-
       list_of_chained_plans(Initial_chained_plans),
       append([[First_chained_plan]],Initial_chained_plans,Final_chained_plans),
       retract(list_of_chained_plans(_)),
```

```
        assert(list_of_chained_plans(Final_chained_plans)),
        update_chained_plans(Remaining_chained_plans).
update_chained_plans([]).

pop_until(Plan) :-
        top_plan(Plan),!.
pop_until(Plan) :-
        plan_stack([Top_plan|Remaining_plans]),
        retract(complete(Top_plan)),
        retractall(last(_,Top_plan)),
        retractall(next(_,Top_plan)),
        retract(plan_stack(_)),
        assert(plan_stack(Remaining_plans)),
        pop_until(Plan).

check_if_all_variables(Arguments) :-
        Arguments = [First_argument|Remaining_arguments],
        var(First_argument),
        check_if_all_variables(Remaining_arguments).
check_if_all_variables([]).

add_to_speech_acts(Speech_act) :-
        possible_speechacts(Initial_speech_acts),
        append(Initial_speech_acts,[Speech_act],Final_speech_acts),
        retract(possible_speechacts(_)),
        assert(possible_speechacts(Final_speech_acts)).

add_to_plans(Plan) :-
        plans(Initial_plans),
        append(Initial_plans,[Plan],Final_plans),
        retract(plans(_)),
        assert(plans(Final_plans)).

push_to_plan_stack(Plans) :-
        Plans=[First_plan|_],
        plan_stack([Top_plan|Remaining_plans]),
        First_plan=Top_plan,
        retract(plan_stack(_)),
        assert(plan_stack([First_plan|Remaining_plans])),!,
        Remaining_plans=[First_of_remaining_plans|_],
        next(Action,First_of_remaining_plans),
        retract(next(_,First_of_remaining_plans)),
        assert(next(Action,First_of_remaining_plans)).

push_to_plan_stack(Plans) :-
        plan_stack(Initial_plan_stack),
        append(Plans,Initial_plan_stack,Final_plan_stack),
        retract(plan_stack(_)),
        assert(plan_stack(Final_plan_stack)).

restart :-
        retract(possible_speechacts(_)),
        assert(possible_speechacts([])),
        retract(plans(_)),
        assert(plans([])),
        retract(list_of_chained_plans(_)),
```

```
        assert(list_of_chained_plans([])).

nottrue([First_effect|Remaining_effects]) :-
        \+ First_effect,
        nottrue(Remaining_effects).
nottrue([]).

top_plan(Top_plan) :-
        plan_stack([Top_plan|_]).

execute([First_effect|Remaining_effects]) :-
        assert(First_effect),
        execute(Remaining_effects).
execute([]).

testforend :-
        top_plan(end_plan(_,_,_)),
        plan_stack(Plan_stack),
        all_complete(Plan_stack).

all_complete([]).
all_complete([First_plan|Remaining_plans]) :-
        complete(First_plan),
        all_complete(Remaining_plans).

get_unknown(Arguments,Unknown) :-
        Arguments = [Unknown|_],
        var(Unknown),!.
get_unknown(Arguments,Unknown) :-
        Arguments = [_|Remaining_arguments],
        get_unknown(Remaining_arguments,Unknown).

check_no_unknowns([First_argument|Remaining_arguments]) :-
        \+(var(First_argument)),
        check_no_unknowns(Remaining_arguments).
check_no_unknowns([]).

perform(Action,Plan) :-
        retract(next(Action,Plan)),
        assert(last(Action,Plan)),
        assert(complete(Plan)),
        write('s : surface_inform(s,u,done(s,u,'),write(Action),write('))'),nl,nl.

/*************************************************************************
                        Discourse Plan Library
                        ----------------------
*************************************************************************/

:- multifile prereqs/2.
:- multifile decomp/2.
:- multifile addlist/2.

/*************************************************************************
                S : Speaker
                H : Hearer
                P : Plan
```

```
              Par : Parameter
                A : Action
              New : New step to be inserted in plan
             Next : Next step in plan which is not executable and hence
                    signals correction in plan
               SA : Speech act
*************************************************************************/

/*************************************************************************
        prereqs(A,B) : B is a list of prerequisites for discourse plan A
         decomp(A,B) : B is a decomposition of discourse plan A
        addlist(A,B) : B is a list of effects that wiil be true after
                       successful completion of discourse plan A
                       [The constraints are included in the bodies of the
                        clauses for decomp]
*************************************************************************/

prereqs(identify_parameter(_,_,_,_,_),[]).
decomp(identify_parameter(S,H,Par,A,P),informref(S,H,Par,parameter(Par,P))) :-
        step(A,P),
        retract(next(_,identify_parameter(_,_,_,_,_))).
addlist(identify_parameter(S,H,Par,A,P),[complete(identify_parameter(S,H,Par,A,P
)),last(informref(S,H,Par,parameter(Par,P)),identify_parameter(S,H,Par,A,P))]).

prereqs(correct_plan(_,_,L,_,_,P),[last(L,P)]).
decomp(correct_plan(S,H,L,New,Next,P),request(S,H,New)) :-
        top_plan(P),
        complete(P),
        last(L,P),
        Next = none,
        retractall(speech_act(_)),
        assert(speech_act(request(S,H,New))).
addlist(correct_plan(S,H,L,New,Next,P),[complete(correct_plan(S,H,L,New,Next,P))
,next(New,P),last(SA,correct_plan(S,H,L,New,Next,P))]) :-
        speech_act(SA).

prereqs(introduce_plan(_,_,_,_),[]).
decomp(introduce_plan(S,H,A,P),request(S,H,A)) :-
        step(A,P),
        retractall(speech_act(_)),
        assert(speech_act(request(S,H,A))).
addlist(introduce_plan(S,H,A,P),[complete(introduce_plan(S,H,A,P)),next(A,P),las
t(SA,introduce_plan(S,H,A,P))]) :-
        speech_act(SA).

prereqs(end_plan(_,_,_),[]).
decomp(end_plan(S,H,P),acknowledge(S,H)) :-
        top_plan(P).
addlist(end_plan(S,H,P),[complete(end_plan(S,H,P)),last(acknowledge(S,H),end_pla
n(S,H,P))]).

/*************************************************************************
                        Domain Plan Library
                        -------------------
*************************************************************************/
:- multifile prereqs/2.
```

```
:- multifile decomp/2.
:- multifile addlist/2.

/*************************************************************************
                S : Speaker
                H : Hearer
                F : Feature
                T : Phone number where calls are forwarded to
             From : Phone number where calls are forwarded from
             PorC : Phone number or a code in the speed-dial directory
                P : Phone number in speed dial
                O : Old code for phone number P
                N : The code to which O has to be changed to
                C : Code which has to be added for phone number P
*************************************************************************/

/*************************************************************************
        prereqs(A,B) : B is a list of prerequisites for domain plan A
         decomp(A,B) : B is a decomposition of domain plan A
        addlist(A,B) : B is a list of effects that will be true after
                       successful completion of domain plan A
*************************************************************************/

prereqs(gethelp(_,_,_),[]).
decomp(gethelp(S,H,F),help(H,S,F)).
addlist(gethelp(S,H,F),[complete(gethelp(S,H,F)),last(help(H,S,F),gethelp(S,H,F))]).
dellist(gethelp(S,H,F),[next(help(H,S,F))]).


prereqs(getcancel(_,_,_),[]).
decomp(getcancel(S,H,F),cancel(H,S,F)).
addlist(getcancel(S,H,F),[complete(getcancel(S,H,F)),last(cancel(H,S,F),getcance
l(S,H,F))]).
dellist(getcancel(S,H,F),[next(cancel(H,S,F))]).


prereqs(getforward(_,_,_),[]).
decomp(getforward(S,H,T),forward(H,S,T)).
addlist(getforward(S,H,T),[complete(getforward(S,H,T)),last(forward(H,S,T),getfo
rward(S,H,T))]).
dellist(getforward(S,H,T),[next(forward(H,S,T))]).


prereqs(getremforward(_,_,_,_),[]).
decomp(remforward(S,H,From,T),remforward(H,S,From,T)).
addlist(getremforward(S,H,From,T),[complete(getremforward(S,H,From,T)),last(remf
orward(H,S,From,T),getremforward(S,H,From,T))]).
dellist(getremforward(S,H,From,T),[next(remforward(H,S,From,T))]).


prereqs(getdial(_,_,_),[]).
decomp(getdial(S,H,PorC),dial(H,S,PorC)).
addlist(getdial(S,H,PorC),[complete(getdial(S,H,PorC)),last(dial(H,S,PorC),getdi
al(S,H,PorC))]).
dellist(getdial(S,H,PorC),[next(dial(H,S,PorC))]).
```

```
prereqs(getchange(_,_,_,_,_),[]).
decomp(getchange(S,H,P,O,N),change(H,S,P,O,N)).
addlist(getchange(S,H,P,O,N),[complete(getchange(S,H,P,O,N)),last(change(H,S,P,O
,N),getchange(S,H,P,O,N))]).
dellist(getchange(S,H,P,O,N),[next(change(H,S,P,O,N))]).


prereqs(getadd(_,_,_,_),[]).
decomp(getadd(S,H,P,C),add(H,S,P,C)).
addlist(getadd(S,H,P,C),[complete(getadd(S,H,P,C)),last(add(H,S,P,C),getadd(S,H,
P,C))]).
dellist(getadd(S,H,P,C),[next(add(H,S,P,C))]).


prereqs(getdelete(_,_,_),[]).
decomp(getdelete(S,H,PorC),delete(H,S,PorC)).
addlist(getdelete(S,H,PorC),[complete(getdelete(S,H,PorC)),last(delete(H,S,PorC)
,getdelete(S,H,PorC))]).
dellist(getdelete(S,H,PorC),[next(delete(H,S,PorC))]).


prereqs(hearcodes(_,_),[]).
decomp(hearcodes(S,H),showcodes(H,S)).
addlist(hearcodes(S,H),[complete(hearcodes(S,H)),last(showcodes(H,S),hearcodes(S
,H))]).
dellist(hearcodes(S,H),[next(showcodes(H,S))]).

/*************************************************************************
                        Speech Act Library
                        ------------------
*************************************************************************/
:- multifile prereqs/2.
:- multifile decomp/2.
:- multifile addlist/2.

/*************************************************************************
                S : Speaker
                H : Hearer
                A : Action
                P : Proposition
                T : Term
*************************************************************************/

/*************************************************************************
        prereqs(A,B) : B is a list of prerequisites for speech act A
         decomp(A,B) : B is a decomposition of speech act A
        addlist(A,B) : B is a list of the effects that will be true
                       after successful completion of speech act A
*************************************************************************/

prereqs(request(_,_,_),[]).
decomp(request(S,H,A),surface_request(S,H,A)).
decomp(request(S,H,A),surface_yn_question(S,H,cando(H,A))).
decomp(request(S,H,A),surface_inform(S,H,not(cando(S,A)))).
decomp(request(S,H,A),surface_inform(S,H,want(S,A))).
```

```
addlist(request(_,_,_),[]).

prereqs(inform(S,_,P),[know(S,P)]).
decomp(inform(S,H,P),surface_inform(S,H,P)).
addlist(inform(S,H,P),[know(H,P),know(H,know(S,P))]).

prereqs(informref(_,_,_,_),[]).
decomp(informref(S,H,T,P),surface_informref(S,H,T,P)) :-
        parameter(T,P).
decomp(informref(S,H,T,P),surface_request(S,H,A)) :-
        A =.. [_,_,_|Z],
        check_if_all_variables(Z),
        functor(A,T,_),
        fw(T),
        P = parameter(T,_).
decomp(informref(S,H,T,P),surface_request(S,H,A)) :-
        A = dial(H,S,T),
        P = parameter(T,_).
addlist(informref(_,_,_,_),[]).

prereqs(informif(_,_,_,_),[]).
decomp(informif(S,H,P,TorF),surface_informif(S,H,P,TorF)).
addlist(informif(_,_,_),[]).

prereqs(yn_question(_,_,_),[]).
decomp(yn_question(S,H,Prop),surface_yn_question(S,H,Prop)) :-
        \+ (know(S,Prop)).

prereqs(acknowledge(_,_),[]).
decomp(acknowledge(S,H),surface_acknowledge(S,H)).
addlist(acknowledge(_,_),[]).

/*************************************************************************
                                Database
                                --------
*************************************************************************/

:- dynamic complete/1.
:- dynamic next/2.
:- dynamic last/2.
:- dynamic speech_act/1.

step(help(H,S,F),gethelp(S,H,F)).
step(cancel(H,S,F),getcancel(S,H,F)).
step(forward(H,S,To),getforward(S,H,To)).
step(remforward(H,S,From,To),getremforward(S,H,From,To)).
step(dial(H,S,N),getdial(S,H,N)).
step(change(H,S,P,O,N),getchange(S,H,P,O,N)).
step(add(H,S,P,C),getadd(S,H,P,C)).
step(delete(H,S,PorC),getdelete(S,H,PorC)).
step(showcodes(H,S),hearcodes(S,H)).

/*************************************************************************
        Initializations ......
*************************************************************************/
complete(none).
```

```
next(none,none).

last(none,none).

speech_act(none).

/*************************************************************************
        fw(A) : A is a feature word
*************************************************************************/
fw(forward).
fw(wait).
fw(dial).
fw(remote).
fw(speed).
```

REFERENCES

Allen, J. F., and C. R. Perrault. "Analyzing intention in utterances." *Artificial Intelligence, 15.3.* (1980): 143-178.

Balentine, J., Berry, U., and A. Johnstone. "The DIM system : WOZ Simulation Results - Phase I." *Technical Report WUCS-92-20, Computer Sciience, Washington University, St. Louis.* (1992).

Balentine, J., Johnstone, A., and D. Mathias. "The DIM system : an Empirically Derived NLP system." *Technical Report WUCS-92-21, Computer Sciience, Washington University, St. Louis.* (1992).

Fikes, R. E., and N. J. Nilsson. "STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence, 2,* (1971): 189-208.

Hinkelman, E. "Linguistic and Pragmatic constraints on Utterance Interpretation." *Ph. D. Thesis and Technical Report 288, University of Rochester, Department of Computer Science.* (1990)

Johnstone, A., and J. Balentine. "Integrating speech recognition and plan-based dialogue processing." *Proceedings from the workshop on Empirical Models and Methodology for Natural Language Dialogue Systems, Third conference on Applied NLP, Trento, Italy* (1992).

Lambert, L., and S. Carberry. "A Tripartite Plan-based model of Dialogue." *Proceedings from Association for Computational Linguistics, 1991,* (1991): 47-54.

Litman, D. J., and J. F. Allen. "A plan recognition model for subdialogues in conversations." *Cognitive Science, 11.2,* (1987): 163-200.

Polanyi, L., and R. J. H. Scha. "The syntax of discourse." *Text, 3,* (1983): 261-270.

Reichman-Adar, R. "Extended person-machine interfaces." *Artificial Intelligence, 22,* (1984): 157-218.

Searle, J. R. "Speech acts, an essay in the philosophy of language." *New York: Cambridge University Press* (1969).

Smith, R.W., Hipp, D. R., and A. W. Biermann. "A Dialog Control Algorithm and its Performance." *Proceedings of the Third Conference on Applied Natural Language Processing, Trento, Italy* (1992): 9-16.

Traum, D. R. "Towards a Computational Theory of Grounding in Natural Language Conversation." *Technical report 401, Computer Science, University of Rochester,* October (1991).